4.5 For Loop Variations

In the last section we introduced for loops and the accumulator pattern. The examples we used all had very similar code, with some differences in the type of collection we iterated over and how we initialized and updated our accumulator variable. In this section, we'll study two variations of the basic loop accumulator pattern: having multiple accumulator variables for the same loop, and using if statements to perform a *conditional update* of loop accumulators.

Before proceeding, please take moment to review the loop accumulator pattern:

```
<x>_so_far = <default_value>

for element in <collection>:
    <x>_so_far = ... <x>_so_far ... element ...  # Somehow combine loop
         variable and accumulator

return <x>_so_far
```

Multiple accumulators

In each example from the last section we used only one accumulator. The pattern can be extended to use multiple accumulators. For example, given a dictionary mapping menu items to prices, how can we get the average price? Remember that an average requires both the sum and the number of elements. We can create two accumulators to accomplish this:

```
def average_menu_price(menu: dict[str, float]) -> float:
    """Return the average price of an item from the menu.

    >>> average_menu_price({'fries': 3.5, 'hamburger': 6.5})
    5.0
    """
    # ACCUMULATOR len_so_far: keep track of the number of
    # items in the menu seen so far in the loop.
    len_so_far = 0
    # ACCUMULATOR total_so_far: keep track of the cost of
    # all items in the menu seen so far in the loop.
    total_so_far = 0.0

    for item in menu:
        len_so_far = len_so_far + 1
        total_so_far = total_so_far + menu[item]

    return total_so_far / len_so_far
```

Here is how we could write a loop accumulation table for this example:

Iteration	Loop variable (item)	Accumulator len_so_far	Accumulator total_so_far
0		0	0.0
1	'fries'	1	6.5
2	'hamburger'	2	10.0

Conditional execution of the accumulator

Consider the following problem: given a string, count the number of vowels in the string.

```
def count_vowels(s: str) -> int:
    """Return the number of vowels in s.

    >>> count_vowels('aeiou')
    5
    >>> count_vowels('David')
    2
    """
```

We saw in 4.4 Repeated Execution: For Loops that we could count *every* character in a given string by using an accumulator that increased by 1 for every loop iteration. We can use the same idea for counting just vowels, but we need to increase the accumulator only when the current character is a vowel.

In Chapter 3, we learned how to control execution of whole blocks of code using if statements. By nesting an if statement inside a for loop, we can adapt our accumulator pattern to only update the accumulator when certain conditions are met.

```
def count_vowels(s: str) -> int:
    """Return the number of vowels in s.

    >>> count_vowels('aeiou')
    5
    >>> count_vowels('David')
    2
    """
    # ACCUMULATOR vowels_so_far: keep track of the number of vowels
    # seen so far in the loop.
    vowels_so_far = 0

    for letter in s:
        if letter in 'aeiou':
```

```
            vowels_so_far = vowels_so_far + 1

    return vowels_so_far
```

If `word` is the empty string, the for loop will not iterate once and the value 0 is returned. This tells us that we have initialized our accumulator correctly. What about the loop body? There are two cases to consider:

1. When `letter` is a vowel, the reassignment `vowels_so_far = vowels_so_far + 1` increases the number of vowels seen so far by 1.
2. When `letter` is not a vowel, nothing else happens in the current iteration because this if statement has no else branch. The vowel count remains the same.

Here's our loop accumulation table for `count_vowels('David')`. At each iteration, the accumulator either stays the same (when `letter` is not a vowel) or increases by 1 (when `letter` is a vowel).

Loop Iteration	Loop Variable `letter`	Accumulator `vowels_so_far`
0		`0`
1	`'D'`	`0`
2	`'a'`	`1`
3	`'v'`	`1`
4	`'i'`	`2`
5	`'d'`	`2`

We can also contrast this function to an equivalent implementation using a filtering comprehension:

```
def count_vowels(s: str) -> int:
    """Return the number of vowels in s.

    >>> count_vowels('aeiou')
    5
    >>> count_vowels('David')
    2
    """
    return len([letter for letter in s if letter in 'aeiou'])
```

This version hopefully makes clear that the `if letter in 'aeiou` in the loop version acts as a *filter* on the string `s`, causing the loop accumulator to only be updated for the vowels. In this version, the actual accumulation (`vowels_so_far = vowels_so_far + 1`) is handled by the call to `len`.

Implementing `max`

Now let's consider implementing another built-in aggregation function: `max`. We'll require that the input be non-empty, as we cannot compute the maximum element of an empty collection. This allows us to set the initial value of our accumulator based on the input.

```
def my_max(numbers: list[int]) -> int:
    """Return the maximum value of the numbers in numbers.

    Preconditions:
        - numbers != []

    >>> my_max([10, 20])
    20
    >>> my_max([-5, -4])
    -4
    """
    # ACCUMULATOR max_so_far: keep track of the maximum value
    # of the elements in numbers seen so far in the loop.
    max_so_far = numbers[0]

    for number in numbers:
        if number > max_so_far:
            max_so_far = number

    return max_so_far
```

Because we can *assume* that the precondition holds when implementing `my_max`, we can access `numbers[0]` to set the initial value of `max_so_far` without worrying about getting an `IndexError`. In the loop, the accumulator `max_so_far` is updated only when a larger number is encountered (`if number > max_so_far`). Note that here, the term *accumulator* diverges from its normal English meaning. At any point during the loop, `max_so_far` is assigned to a single list element, not some "accumulation" of all list elements see so far. Instead, `max_so_far` represents the *maximum of the elements seen so far*, and so what is being accumulated is a set of facts: "the elements seen so far all `<= max_so_far`".

Existential search

In 3.2 Predicate Logic, we saw how to use `any` to check whether there exists a string in a collection that starts with the letter `'D'`:

```
def starts_with(strings: Iterable[str], char: str) -> bool:
    """Return whether one of the given strings starts with the character
         char.

    Precondition:
        - all({s != '' for s in strings})
        - len(char) == 1
```

```
    >>> starts_with(['Hello', 'Goodbye', 'David', 'Dario'], 'D')
    True
    >>> starts_with(['Hello', 'Goodbye', 'David', 'Dario'], 'A')
    False
    """
    return any({s[0] == char for s in words})
```

Our next goal is to implement this function *without* using the `any` function, replacing it for loops and if statements. If we take a look at the argument to `any` above, we see some pretty big hints on how to do this:

1. The syntax `for s in words` can be used to create a for loop.
2. The expression `s[0] == char` can be used as a condition for an if statement.

Let's give it a shot using our existing accumulator pattern. Because the result of the function is a `bool`, our accumulator will also be a `bool`. Its initial value will be `False`, which is the correct return value when `strings` is empty.

```
def starts_with_v2(words: list[str], char: str) -> bool:
    """..."""
    # ACCUMULATOR starts_with_so_far: keep track of whether
    # any of the words seen by the loop so far starts with char.
    starts_with_so_far = False

    for s in words:
        ...

    return starts_with_so_far
```

How do we update the accumulator? We set it to `True` when the current string `s` starts with `char`, which is exactly the condition from the comprehension.

```
def starts_with_v2(strings: Iterable[str], char: str) -> bool:
    """..."""
    # ACCUMULATOR starts_with_so_far: keep track of whether
    # any of the strings seen by the loop so far starts with char.
    starts_with_so_far = False

    for s in strings:
        if s[0] == char:
            starts_with_so_far = True

    return starts_with_so_far
```

Here is a loop accumulation table for `starts_with(['Hello', 'Goodbye', 'David', 'Mario'], 'D')`. The third iteration assigns `starts_with_so_far` to `True`, while in the other

iterations nothing occurs.

Iteration	Loop variable `s`	Accumulator `starts_with_so_far`
0		`False`
1	`'Hello'`	`False`
2	`'Goodbye'`	`False`
3	`'David'`	`True`
4	`'Mario'`	`True`

Early returns

The function `starts_with_v2` is correct and fits our accumulator pattern well. But you might have noticed that it performs unnecessary work because it must loop through every element of the collection before returning a result. Why is this unnecessary? Because we are interested only in whether *there exists* a string that starts with the given letter! As soon as the condition `s[0] == char` evaluates to `True`, we know that the answer is *Yes* without checking any of the remaining strings.

So the question is, how do we take advantage of this observation to make our code more efficient? We can use a return statement inside the body of the loop. Let's revisit how we described the execution of a return statement in Chapter 2 (new emphasis in **bold**):

> When a return statement is executed, the following happens:
>
> 1. The `<expression>` is evaluated, producing a value.
> 2. That value is then returned to wherever the function was called. **No more code in the function body is executed after this point.**"

In all our functions so far, we have written return statements only at the end of our function bodies or branches of an if statement. This should make sense based on the behaviour described above: any code after a return statement will not execute!

```
return 5
x = 10  # This statement doesn't execute!
```

But we can combine return statements with if statements to conditionally stop executing any more code in the function body. This is called *short-circuiting* or *early returning*.

So our first attempt at making a more efficient `starts_with` is to use an early return inside the if branch:

```
def starts_with_v3(strings: Iterable[str], char: str) -> bool:
    """..."""
```

```
    for s in strings:
        if s[0] == char:
            return True
```

This for loop is strange: it seems we no longer have an accumulator variable! This is actually fairly common for functions that return booleans. Rather than accumulating a `True`/`False` value, it is often possible to directly return the literals `True` or `False`.

The `starts_with_v3` implementation does successfully return `True` on our first doctest example during the *third* loop iteration (when `s = 'David'`), skipping the fourth iteration. However, this implementation will fail the second doctest example (when there are no strings that start with the given character in the collection). We have not explicitly stated what to return when *none* of the strings in `words` starts with `char`. Actually, we have *violated our own type contract* because the function will implicitly return `None` in this scenario.

To fix it, we need to specify what to return if the loop stops without retuning early—this occurs only when there are no strings that start with the given character, and so we return `False`.

```
def starts_with_v4(strings: Iterable[str], char: str) -> bool:
    """..."""
    for s in strings:
        if s[0] == char:
            return True

    return False
```

One common error

When working with early returns inside loops, students often have a tendency to write symmetric if-else branches, like the following:

```
def starts_with_v5(strings: Iterable[str], char: str) -> bool:
    """..."""
    for s in strings:
        if s[0] == char:
            return True
        else:
            return False
```

Unfortunately, while we emphasized symmetry earlier when writing functions with if statements, here symmetry is *not* desirable! With both the if and else branches containing an early return, the loop will only ever perform one iteration. That is, `starts_with_v5` makes a decision about whether to return `True` or `False` just by examining the first string in the collection, regardless of what the other strings are. So if we consider

`starts_with_v5(['Hello', 'Goodbye', 'David', 'Mario'], 'D')`, the only string to be visited in the loop is `'Hello'`, and `False` would be returned!

The lesson here is that existential searches are fundamentally asymmetric: your function can return `True` early as soon as it has found an element of the collection meeting the desired criterion, but to return `False` it must check *every* element of the collection.

Universal search

Now let's consider a dual problem to the previous one: given a collection of strings and a character, return whether *all* strings in the collect start with that letter. If we use the comprehension version of `starts_with`, this change is as simple as swapping the `any` for `all`:

```
def all_start_with(strings: Iterable[str], char: str) -> bool:
    """Return whether all of the given strings starts with the character
         char.

    Precondition:
        - all({s != '' for s in strings})
        - len(char) == 1

    >>> all_starts_with(['Hello', 'Goodbye', 'David', 'Dario'], 'D')
    False
    >>> all_starts_with(['Drip', 'Drop', 'Dangle'], 'D')
    True
    """
    return all({s[0] == char for s in strings})
```

We can also use the accumulator pattern from `starts_with_v2` to check every string. Now, our accumulator starts with the default value of `True`, and changes to `False` when the loop encounters a string that does *not* start with the given letter.[1]

[1] Such a string acts as a *counterexample* to the statement "every string starts with the given character".

```
def all_starts_with_v2(strings: Iterable[str], char: str) -> bool:
    """..."""
    # ACCUMULATOR starts_with_so_far: keep track of whether
    # all of the strings seen by the loop so far starts with char.
    starts_with_so_far = True

    for s in strings:
        if s[0] != char:
            starts_with_so_far = False

    return starts_with_so_far
```

And as before, we can also write this function using an early return, since we can return `False` as soon as a counterexample is found:

```
def all_starts_with_v3(strings: Iterable[str], char: str) -> bool:
    """..."""
    for s in words:
        if s[0] != char:
            return False

    return True
```

Note that this code is very similar to `starts_with_v4`, except the condition has been negated and the `True` and `False` swapped. Existential and universal search are very closely related, and this is borne out by the similarities in these two functions. However, this also illustrates the fact that loops are more complex than using built-in functions and comprehensions: before, we could just swap `any` for `all`, but with loops we have to change a few different areas of the code to make this change.